Faster Computing through Software-Controlled Cache Replacement

Kristof Beyls

Abstract— **Computer program execution speed is determined by the speed of the processor and the memory. The memory is much slower than the processor, leading to an imbalance. To obtain higher performance, it is necessary to eliminate the effect of the slow memory. The traditional solution to this problem is using cache, a small and fast memory containing the most commonly used data. However, the speed gap between the processor and the memory is increasing, and even in the presence of caches, typically the execution speed is halved because of slow accesses to memory. In the presented work, an automated method is devised which allows the cache to be more effective, resulting in speed ups of up to 36%.**

Keywords— **compiler optimization, high performance computing, cache**

I. Introduction

THE two most important components in a computer are the *processor* and the *memory*. The execution speed of computer programs is highly determined by these two components. Processor speed and memory speed improve at different rates. Moore's law says that processor speed improves by 60% per year. Memory speed improves only by 7% per year. This leads to the anti-law of Moore: the speed gap between the memory and the processor doubles every 21 months. Currently, a processor can perform up to 1000 calculations in the time needed to fetch a data element from the memory.

Techniques to resolve the unbalanced computers with fast processors and slow memory, in which the processors are idle most of the time because memory cannot serve data at adequate speed, need to be developed. Caches have been used for more than 20 years to bridge the speed gap between fast processors and slow memories. However, the speed gap between processor and memory has grown so big that traditional caches alone are not enough anymore. Therefore, in new processors, the cache hardware is made visible to the program. In this way, the program itself can decide which data is kept in the cache.

II. Caches

The laws of nature prohibit constructing large memories with the same speed as the processor. Only small memories at processor speed are feasible. However, most programs need more memory than the amount provided by small and fast memories. This problem can be solved thanks to the principle of locality. This principle says that data that is used by the program will probably be reused in the near future. Caches are small and fast memories which exploit the principle of locality to improve the average memory access time. Typically a current processor has multiple cache levels, where each level has a different tradeoff between size and speed, e.g. a first level cache (L1) of 16KB with the same speed as the processor, a second level cache (L2) of 512KB which is ten times slower, and a main memory of 512MB which is a hundred times slower than the processor.

K. Beyls is with the Department of Electronics and Information Systems, Ghent University (RUG), Gent, Belgium. E-mail: kristof.beyls@elis.rug.ac.be .

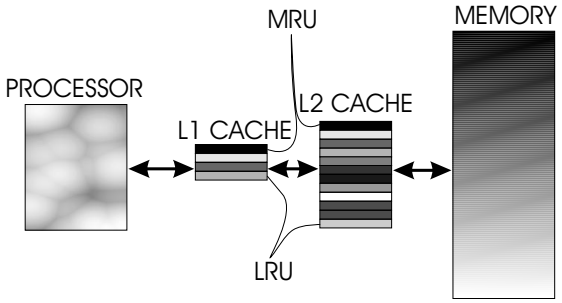

Fig. 1. Multiple levels of cache are located between the processor and the memory. The most recently used (MRU) data is logically placed at the top of the cache. The least recently used (LRU) data is at the bottom. When data in the cache is used, it is moved to the top of the cache.

A. How Caches Work

When the processor needs data to be processed, it asks the first level of cache to give it the data. If the data is present in the cache (a cache hit), it is quickly returned to the processor. If it is not present (a cache miss), it needs to be fetched from the next level in the cache hierarchy. Upon a cache miss, when the requested data, which is fetched from a lower cache level (or the main memory), is delivered, it must be stored in the cache. However, the cache is small and some other data must be evicted so that there is room for the new data to reside in. Based on the principle of locality, many cache designs choose to replace the data which has not been used for the longest time, since that is the data least likely to be needed in the short future.

The decision about which data to evict from the cache is taken by the cache hardware. In order to remember which data was accessed least recently, the data in the cache is conceptually placed on a stack (see fig. 1). Whenever data in the cache is accessed, it is placed on top of the stack. So the most recently accessed data is at the top of the stack, while the least recently accessed data is at the bottom of the stack. Therefore, whenever data needs to be replaced in the cache, the least recently accessed object can quickly be found at the bottom.

Although caches improve program performance enormously, for a lot of program executions, the processor waits

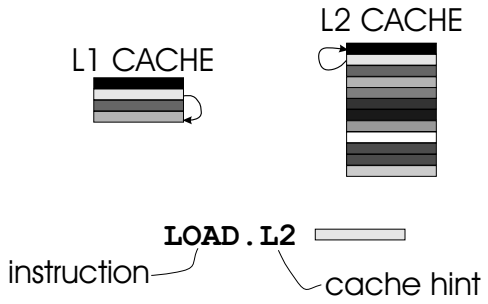

Fig. 2. An example of a cache hint which is annotated to a load instruction. The cache hint L2 indicates that the data should be retained only in the L2 cache and lower level caches, while it should be replaced in the L1 cache. Therefore, in the L1 cache, the requested data is moved to the bottom of the stack, so that it will be the first candidate for replacement. In the L2 cache, the data is moved to the top of the stack, so that the data will be retained for a long time.

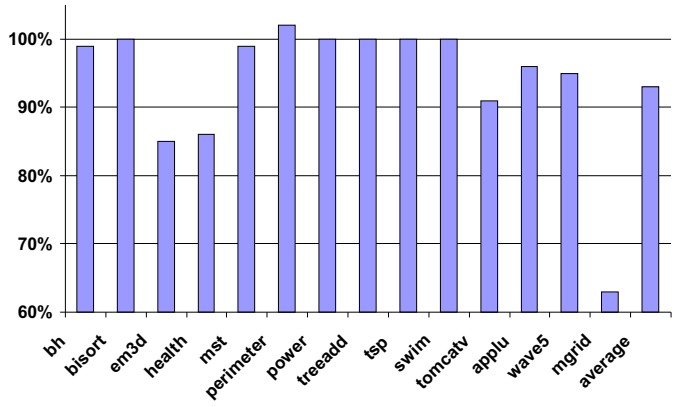

Fig. 3. Relative execution time after cache hint selection. 100%=execution time of the program without cache hints.

for data for more than 50% of the time.

III. Software-Controlled Replacement

Since the speed gap between processor and memory is growing exponentially, ever more powerful techniques are needed make the cache more effective. One of the new possibilities in recent processors is to steer the replacement decision in the cache from the software, i.e. in an instruction which fetches data from the cache, it can be specified whether that data should be retained in that cache or not. The part of the instruction which tells in which cache levels data should be retained is called a *cache hint*. In this work, cache hints are selected so that the caches work more effectively (= there are less cache misses).

A. How Cache Hints Work

A cache hint is annotated to an instruction which accesses the memory. An example can be seen in fig. 2. The hint indicates up to which cache level data should be retained. For the cache levels above the indicated level, the data should not be kept. Therefore, if the cache hint indicates a cache level larger than the cache, the data is moved to the bottom of the stack. If it indicates an equal or larger cache level, the default of moving data to the top of the stack is performed.

B. Cache Hint Selection

Without cache hints, the cache always moves data to the top of the stack when it is accessed. Therefore, before that data is removed from the cache, many accesses to other data are needed. However, if the accessed data is not used in the near future, it might be useful not to retain it in the cache. If it is not retained, cache space becomes available for other data which might be used in the nearer future.

In our cache hint selection scheme, it is examined whether it is useful to keep the data in the cache or not. If the data is accessed so long in the future that it will be evicted from the cache before it is reaccessed, it was not useful to keep it. The second access to the same data will be a cache miss, whether it is retained in the cache or not. If the cache hint indicates that the data should not be retained, it can quickly be removed the next time new data enters the cache. In this way, cache locations become available for data which does exhibit locality and which can profit from being located in the cache.

IV. Evaluation

The cache hint selection scheme described in the previous section has been automated. The program for which cache hints should be annotated to the instructions is first instrumented, so that for every memory access it can be measured how far in the future the requested data will be reaccessed. Then the instrumented program is executed in order to measure the locality for all the data accesses. Based upon this measurement, cache hints are annotated to the memory instructions in the original program. This automated approach has been applied to a number of typical programs. The relative execution time of the programs after cache hint selection is shown in fig. 3. From the figure, it can be seen that the execution time is reduced by 7% on average, with a maximum of 36%.

V. Conclusions

Cache hints and their automated selection has been described as one remedy to the imbalance between the slow memory and the fast processor in current and future computer systems. Carefully selecting cache hints allows to have less cache misses, which results in faster programs. More details about the cache hint selection scheme and its implementation can be found in [1].

Acknowledgments

This research was supported by the Flemish Institute for promotion of scientific and technological research in the industry (IWT).

References

[1] Kristof Beyls and Erik D'Hollander, "Reuse distance-based cache hint selection," in *Proceedings of the 8th International Euro-Par Conference*, 2002, pp. 265–274.